Axiomatization of Database Transformations

Qing Wang and Klaus-Dieter Schewe

Information Science Research Centre
Massey University, Private Bag 11222, Palmerston North, New Zealand
[q.q.wang|k.d.schewe]@massey.ac.nz

Abstract. In database theory, we seek a general computation model for database transformations as an umbrella for queries and updates. However, the literature shows that queries and updates have different flavours, and the completeness standards for query languages cannot be naturally extended to updates. This motivates the question whether we can use ASMs to characterize database transformations over complex value structures in a unified computation model. In this paper we start examining the differences between database transformations and algorithms that give rise to the notion of Abstract Database Transformation Machine (ADTM), which captures computations involved in database transformations over complex value structures, and encompasses not only queries but also updates. We show that ADTMs can behaviourally simulate any database transformation over complex value structures.

1 Introduction

In the literature the notion of database transformation as a formalisation of certain binary relations on database instances encompassing queries and updates first appeared in [3]. Since then a lot of research effort has been investigated to characterisations of different classes of database transformations [1, 17, 20, 18]. In particular, database transformations should satisfy criteria such as well-typedness, effective computability, genericity and functionality, etc. as discussed in [1]. However, the results of these investigations were fruitful only in the application of queries. Extending these results from queries to updates has led to several problems as mentioned in [19]. Although database transformations are defined to consider not only queries but also updates, there does not exist a general characterisation for updates and queries as a whole in database theory.

Abstract State Machines (ASMs) have been accepted as a universal computation model that formalizes the notion of "algorithm". In particular, the sequential Abstract State Machine (ASM) thesis [13] capturing sequential algorithms sheds lights on the study towards a general computation model of database transformations. The question we raise in this paper is, how we can use ASMs to characterize database transformations over complex value structures. To answer this question, we first check what are significant differences between database transformations and general algorithms. Although database transformations provide a special kind of algorithms, further exploration reveals that the differences lead to the necessity of separate considerations.

We start with checking database transformations against the postulates of sequential algorithms defined in [13]. The sequential accessibility principle of the bounded exploration postulate implicitly suggests that states associated with a sequential algorithm are significantly determined by the algorithm itself in the sense that as long as a sequential algorithm is provided, the bounded-exploration witness which is the non-trivial part of these states can be easily figured out. Every element of a state involved in a sequential algorithm is referred to by a ground term. In contrast, the abstract nature of database transformations leads to only structural properties captured by non-ground terms in considerations. Obviously, there is a fundamental mismatch between database transformations and sequential algorithms, which is exactly the same as the mismatch between the hardness of database queries and their Turing complexity justified in [4]. Therefore, database transformations cannot be simply considered as a special kind of sequential algorithms. Instead of the bounded exploration postulate, which only considers how algorithms determine the bounds of computations, we investigate how database transformation programs and states interactively influence on the upper boundary of computational complexity in the exploration-boundary postulate of database transformations.

How about database transformations and parallel algorithms? In [7] a computation model of parallel algorithms was proposed, which indeed describes a class of database transformations over relational structures. However, in the context of complex value structures, database transformations do not always satisfy the postulates of parallel algorithms presented in [6]. More specifically, while relational structures can be captured by the first-order case, complex value structures can be described by higher-order structures in a more natural sense. Although it is feasible to encode higher-order structures with first-order structures, such an encoding may not only lead to undesirable complexity of computations, but also result in an ambiguous understanding of transformations. The immediate consequence of taking into account higher-order structures at states leads to the fact that atomicity of locations cannot be automatically guaranteed. Therefore, the update postulate of parallel algorithms proposed on the basis of first-order structures needs to be re-examined. The study result provides insight into the separate-update postulate for database transformations over complex value structures.

As mentioned before, in database theory any database transformations must respect the genericity principle, which means that only structural properties can be dealt with by database transformations. Consequently, indistinguishable structures that have exactly the same properties may exist in states, and furthermore they must be treated uniformly during database transformations. In general, database transformations permit non-determinism, which provides an approach to separate indistinguishable structures of states. Nevertheless, such non-determinism should not be in control of database transformations, and also not stick to any specified mechanism. The fair treatment for elements of a class of indistinguishable structures leads to the equivalent structure postulate for

database transformations, which provides the restriction of expressive power for database transformations.

To avoid ambiguity it should be necessary to emphasize the assumptions imposed on database transformations in this paper. Firstly, we only consider database transformations with finite runs, which are reflected by the finite sequence postulate. Secondly, database transformations are considered in the context of complex value databases, which lead to the abstract state postulate being modified to capture higher-order structures.

The remainder of the paper is organized as follows. In Section 2 we present the postulates for database transformations. Then the notion of Abstract Database Transformation Machine is developed in Section 3. Section 4 sketches the proof for the main result obtained in this paper that each database transformation can be simulated by an Abstract Database Transformation Machine. Finally, we summarize this paper in Section 5.

2 Postulates of Database Transformations

In this section we stipulate several postulates that any database transformation must cope with. Each postulate will be accompanied by justifications.

Definition 1. *A database transformation is an object satisfying the finite sequence, abstract state, data abstraction, exploration boundary, separate update and equivalent structure postulates.*

Before presenting the postulates in detail, we briefly comment on the purposes which the postulates serve. The finite sequence postulate states that only database transformations with finite runs are of interest. The abstract state postulate claims that the underlying states have higher-order structures. The data abstraction postulate specifies that database transformations can be executed over complex value structures with flexible build-in type systems. The exploration boundary and equivalent structure postulates aim at figuring out the upper boundaries of complexity and expressive power during one-step transformations, respectively, while the separate update postulate clarifies how complex value structures can be manipulated during one-step transformations.

2.1 Finite Sequence Postulate

The non-deterministic sequential time postulate defined for bounded-choice sequential algorithms in [14] can be directly adapted to be the finite sequence postulate for database transformations with some minor changes. Initial and final states are emphasized in the finite sequence postulate to reflect input and output databases associated with database transformations.

Definition 2. ***(finite sequence postulate).*** *A database transformation Π is associated with a tuple (S_Π, τ_Π), where*

– S_Π *is a non-empty set of states with a distinguished initial state* $s_0 \in S_\Pi$ *and a subset* F_Π *of final states, and*
– τ_Π *is a one-step transition relation over* S_Π*, i.e.* $\tau_\Pi \subseteq S_\Pi \times S_\Pi$.

Definition 3. *A* run *of* $\Pi = (S_\Pi, \tau_\Pi)$ *is a finite sequence of states*

$$s_0, s_1, \ldots, s_n$$

where s_0 *is the initial state of* Π *and* $(s_i, s_{i+1}) \in \tau_\Pi$ $(i \in [0, n-1])$. *A run is* terminated *at a final state, i.e.* $s_n \in F_\Pi$. *The power of the relation* τ_Π^n *satisfying* $(s_0, s_n) \in \tau_\Pi^n$ *is called the* length *of that run.*

Definition 4. *The* run relation *of* $\Pi = (S_\Pi, \tau_\Pi)$ *is the set of all runs of* Π*, and its* arity *is the maximum among lengths of the runs.*

Definition 5. *Let* $\Pi = (S_\Pi, \tau_\Pi)$ *and* $\Pi^{'} = (S_{\Pi'}, \tau_{\Pi'})$ *be two database transformations, then* Π *and* Π' *are* behaviourally equivalent *iff* $S_\Pi = S_{\Pi'}$, $s_0 = s_0^{'}$, $F_\Pi = F_{\Pi'}$ *and* $\tau_\Pi = \tau_{\Pi'}$*, where* $s_0 \in S_\Pi, s_0^{'} \in S_{\Pi'}$ *are initial states of* Π *and* $\Pi^{'}$*, respectively, and* $F_\Pi \subseteq S_\Pi, F_{\Pi'} \subseteq S_{\Pi'}$ *are sets of final states of* Π *and* $\Pi^{'}$*, respectively.*

Note that the above definitions have been defined on database transformations instead of database transformation programs. The difference lies in the fact that a database transformation program should be able to be executed over all reasonable states with various structures, instead of a specific set of states. Here reasonable states refer to any state, on which a database transformation program makes sense. Nevertheless, database transformations are only concerned about behavioural sequences of states.

The following fact is obvious according to the definition of behavioural equivalence of two database transformations.

Fact 1 *Two behaviourally equivalent database transformations must have the same run relation.*

2.2 Abstract State Postulate

To respect conventions adopted in database theory, we revised the abstract state postulate of algorithms presented in [13], and developed the abstract state postulate for database transformations as follows.

Definition 6. ***(abstract state postulate).*** *Let* $\Pi = (S_\Pi, \tau_\Pi)$ *be a database transformation, then*

– *all states in* S_Π *are higher-order structures, and closed under isomorphisms,*
– *each state* $s \in S_\Pi$ *is associated with the same infinite base set* $\mathcal{A}$*, and may have different finite active domains* $Adom(s) \subseteq \mathcal{A}$*,*
– *the signature of* Π *is fixed and finite, consisting of state signature* V_S *and background signature* V_K*, and*

– *for any isomorphism h from state s_1 onto state $h(s_1)$, if $(s_1, s_2) \in \tau_\Pi$, then $(h(s_1), h(s_2)) \in \tau_\Pi$.*

States of database transformations are considered to be higher-order structures, instead of first-order structures as in ASMs in general [13]. Although it has been known that higher-order structures can be mapped to first-order structures, higher-order structures are more appropriate and natural than first-order structures in the context of database transformations over complex value structures. In connection with this point, the most straightforward example is query languages over tree-based structures. As discussed in [15], monadic second-order logic is studied as being logic formalizations for query languages over tree-based structures as an analogy of first-order logic for relational structures. Even for database transformations over relational structures, views as computable queries are also beyond first-order structures if we treat tables as first-order structures in a natural sense.

In terms of a database transformation, there is an unchanged base set $\mathcal{A}$ associated with its all states. More precisely, the base set $\mathcal{A}$ of a state s consists of an infinite dispose domain $Ddom(s)$, a finite active domain $Adom(s)$ and an infinite reserve domain $Rdom(s)$, i.e. $\mathcal{A} = Ddom(s) \cup Adom(s) \cup Rdom(s)$. Furthermore, these domains are pairwise disjoint. Although the base set $\mathcal{A}$ does not change during a database transformation, some of elements can alter among the domains in accordance with a unidirectional flow.

$$dispose \Leftarrow active \Leftarrow reserve$$

The purpose of specifying an infinite dispose domain as part of a base set is to deal with elements that have the identification property, such as object identifiers widely used in the object-oriented paradigm. The unidirectional alteration among dispose, active and reserve domains is critical to preserve the desirable properties of database transformations under composition. Further details regarding this can refer to [19].

In the classical database theory it is common to consider a database transformation as a binary relation over database instances with possibly different schemata. Should we specify alterable state signatures for database transformations to faithfully reflect possible changes on database schemata? By analyzing the factors leading to such changes we believe that states of a database transformation being constituted by not only the underlying databases, but also information from the associated database transformation programs would be more reasonable than states simply being databases. Consequently, state signatures are fixed during a database transformation due to the fact that any changes on database schemata have to be explicitly introduced by database transformation programs. Such a state signature V_S consists of a finite, non-empty set $Fun(V_S)$ of function names associated with fixed arities. In particular, there is a subset $Pre(V_S)$ of function names, called predicate names. Moreover, any state signature must contain a set of logical function names, including logical constants $true$ and $false$, logical connectives such as $\vee$, $\wedge$, etc. and a special symbol $\perp$

as usual. There are several ways to categorize function names. According to the tradition of ASMs, function names can be static and dynamic. Since database transformations include updates, all predicate names are dynamic. In addition, function name can be persistent and temporary. Temporary function names are local names in the sense that all temporary function names will disappear when a database transformation terminates. In other words, no temporary function names can be shared by any two runs, which implicitly states no information carried by temporary functions can be passed on to other runs. Obviously, persistent predicate names are global names, and it is possible to share them within multiple runs.

In terms of a database transformation, the background signature V_K and state signature V_S are always disjoint. However, the union of them constitutes the entire signature of a database transformation. The details about a background signature V_K will be discussed separately in the data abstraction postulate.

The last condition in the abstract state postulate is dealing with the genericity principle of database transformations, which guarantees that the one-step transition relation only handles structure properties instead of explaining atomic value. Since a state can be enlarged by introducing new elements or shrink by eliminating existing elements during a one-step transition, such an isomorphism is defined over an inalterable base set.

2.3 Data Abstraction Postulate

The data abstraction postulate is developed on the basis of the background postulate of [6] with a slightly modified concern. After examining various data models in the literature [11, 10, 5, 16, 21] we observe that it is a common practice to use structure data types, such as set, list, bag, record, union, etc. to reflect a better understanding of complex value structures, no matter which other principles the data models adopt. The data abstraction postulate focuses on capturing this observation by abstracting from any concrete data types.

Definition 7. ***(data abstraction postulate).*** *For a database transformation $\Pi = (S_\Pi, \tau_\Pi)$, a* background *$\mathcal{K}$ for states of S_Π must satisfy the following:*

- *The background signature V_K contains a finite (possibly empty) set T_K of type symbols that are unary function names, a finite (possibly empty) set O_K of constructor symbols.*
- *Type and constructor symbols are classified into a set G of groups.*
- *A background with $T_K = \emptyset$ and $O_K = \emptyset$ is called* transparent.

In terms of a state s with the base set $\mathcal{A}$ and the background $\mathcal{K}$, there exists a set of background structures, denoted by $\mathcal{K}(\mathcal{A})$, obtained by applying background $\mathcal{K}$ over $\mathcal{A}$ as follows:

Let $G = \{g_1, ..., g_n\}$ be the set of groups associated with the background $\mathcal{K}$ such that each group g_i ($i \in [1, n]$) contains a set T_i of type symbols and a set O_i of constructor symbols satisfying $\bigcup_{1 \le i \le n} T_i = T_K$ and $\bigcup_{1 \le i \le n} O_i = O_K$, then

$$\mathcal{K}(\mathcal{A})=\bigcup_{1\leq i\leq n} g_i(\mathcal{A}),$$

where $g_i(\mathcal{A})$ is constructed by applying the following rules:

- For any $t \in T_i$, $t(a) \in g_i(\mathcal{A})$, where $a \in \mathcal{A}$.
- For any $o \in O_i$ and every natural number m, $o(a_1, ..., a_m) \in g_i(\mathcal{A})$, where $a_j \in \mathcal{A}$ ($j \in [1, m]$).
- For any $t \in T_i$, $t(b) \in g_i(\mathcal{A})$, where $b \in g_i(\mathcal{A})$.
- For any $o \in O_i$ and every natural number m, $o(b_1, ..., b_m) \in g_i(\mathcal{A})$, where $b_j \in g_i(\mathcal{A})$ ($j \in [1, m]$).

The classification of type and constructor symbols provides a capability for building up stratified type systems. Within a group, its type and constructor symbols can recursively build structure values over atomic values in a base set, whereas type and constructor symbols in different groups are not allowed to apply on each other. For example, given a state with a base set $\mathcal{A}$ containing elements in $\{21, 12, \text{“}Joe\text{”}, \text{“}Bob\text{”}\}$, and a background $\mathcal{K}$ including a set of type symbols $T_K = \{SCORE, FNAME\}$ and a set of constructor symbols $O_K = \{list, set\}$, if there exists two groups $(SCORE, set)$ and $(FNAME, list)$, then $list(FNAME(\text{“}Joe\text{”}), FNAME(\text{“}Bob\text{”}))$ and $SCORE(set(21, 12))$ are valid background structures of $\mathcal{K}(\mathcal{A})$. However, $SCORE(list(21, 12))$ is not legal because $SCORE$ and *list* belong to different groups.

The structure Y of such a state is constituted by a base set, a set of background structures and the interpretation of function names in the state signature over the base set and background structures. More precisely, for a state s of database transformations, let $\mathcal{B}= \mathcal{A} \cup \mathcal{K}(\mathcal{A})$ represent its *complex value set* containing all atomic values in the base set $\mathcal{A}$ and all structure values in $\mathcal{K}(\mathcal{A})$, then any function with name $f \in Fun(V_S)$ and arity n is a mapping $f : \mathcal{B}^n \rightarrow \mathcal{B}$, while each predicate with name $p \in Pre(V_S)$ and arity n is a mapping $p : \mathcal{B}^n \rightarrow \{true, false\}$. Two states s_1 and s_2 is said to be identical, denoted by $s_1 = s_2$, iff their structures are identical. Apparently, the structure of a state with a transparent background is reduced to be a base set $\mathcal{A}$ and the interpretation of function names in the state signature over $\mathcal{A}$.

Although the set $\mathcal{B}$ is infinite, there is only a finite subset B of $\mathcal{B}$ involved in computations at each state. For convenience, we use the notation $|B|$ to indicate the cardinality of B.

In fact, the data abstraction postulate is a specialization of the background postulate by applying the general principle behind the background postulate, which is to introduce all the information relevant to the future progress of that computation into a state [6], in the context of data modelling. Therefore, the data abstraction postulate provides a general treatment to assemble and dissemble data towards different abstractions of interest.

2.4 Exploration Boundary Postulate

Different from sequential algorithms, parallel computations are involved in database transformations due to evaluations of non-ground terms on states. Although for

such an evaluation the number of the resulting parallel computations only depends on the state, the number of evaluations occurred within a one-step transformation is determined by the associated program. Therefore, our interest here is to investigate the upper boundary of exploration with respect to one-step transformations by taking into account both database transformation programs and states.

This investigation proceeds in two steps. Firstly, the upper boundary of exploration over a state with a transparent background is discussed. Then, the effects of a non-trivial background on the exploration of a state are identified.

For a database transformation, any program to generate a one-step transition relation can contain only a finite set of distinct variables. Here we only consider variables in a mathematical sense. Moreover, the number of distinct variables in a program can be minimized by reusing variables as much as possible provided that the semantics of the program does not change. We call this minimal number n of distinct variables *boundary power*. In addition, since the number of parallel computations is only determined by the underlying state, which is assumed to have a transparent background at this moment, the maximal number of parallelism caused by each variable should be equivalent to the cardinality of the active domain of the state, i.e. $|Adom(s)|$. We call this number *boundary base*. Why we exclude dispose and reserve domains from the boundary base? A database is constructive, new elements imported in a batch must depend on existing elements in the active domain, otherwise, new elements have to be imported one-by-one. The same reason serves for the dispose domain. Based on the above discussions, it can be concluded that there exists an upper boundary $\mathcal{O}(|Adom(s)|^n)$ of exploration over a state with a transparent background during a one-step transformation.

Now let us examine the upper boundary of exploration over a state with a non-trivial background to see how a background influences on the exploration of a database transformation. As said before, there always exists a finite subset B of the complex value set $\mathcal{B}$ involved in a one-step transformation. The involved elements can be either atomic values or structure values. Therefore, the boundary base under a state with a non-trivial background should be $|B|$, instead of $|Adom(s)|$, while the boundary power will remain the same due to the identity of a program. This leads straightforwardly to the following postulate concerning the upper boundary of exploration.

Definition 8. ***(exploration boundary postulate).*** *Let $\Pi = (S_\Pi, \tau_\Pi)$ be a database transformation, then for any state $s \in S_\Pi$ with a base set $\mathcal{A}$ and a background $\mathcal{K}$, there exists two finite numbers n and m such that*

- *the upper boundary of exploration for τ_Π is $\mathcal{O}(n^m)$,*

  where n depends on ($\mathcal{A}$, $\mathcal{K}$), and m depends on τ_Π.

2.5 Separate Update Postulate

Before presenting the separate update postulate, we need to introduce a new notion called *location operator* in advance. Indeed, location operators can be

treated as a special kind of aggregation functions defined in [12], and thus we present the definition of location operator in a similar style. For clarity, we use the notations $\{\!\!\{\}\!\!\}$ for *multiset* and $\uplus$ for function *multiset sum* defined as $multiset \times multiset \rightarrow multiset$ with the obvious meaning.

Definition 9. *Let D be a domain, $\mathcal{M}(D)$ be the set of all non-empty multisets over D, and ρ be a location operator over $\mathcal{M}(D)$, then ρ is a triple $(\alpha, \odot, \beta)$, where*

- $\alpha : D \rightarrow D$ *is a unary function,*
- $\odot$ *is a commutative and associative binary operation over D, and*
- $\beta : D \rightarrow D$ *is a unary function.*

Furthermore, the following condition must be satisfied for $m \in \mathcal{M}(D)$ and $m = \{\!\!\{b_1, ..., b_n\}\!\!\}$ such that

$$\rho(m) = \beta(\alpha(b_1) \odot \cdots \odot \alpha(b_n)).$$

In our work, updates and update sets are defined as usual as in ASMs. To deal with updates occurred in parallel computations, the notion of *update multiset* is indispensable, which has been sufficiently justified in [6].

Definition 10. *For a state s with the structure Y, let f be a n-arity dynamic function name in its state signature, $a_1, ..., a_n$ and b be elements of Y, then an update is a pair*

$$(f(a_1, ..., a_n), b),$$

where $f(a_1, ..., a_n)$ is a location, b is an update value, and the value of $f(a_1, ..., a_n)$ in structure Y, denoted by $f^Y(a_1, ..., a_n)$, is the location content. An update set is a set of updates, while an update multiset is an unordered collection of updates, which allows duplicates.

Let $\mathcal{L}$ and $\mathcal{P}$ be sets of locations and location operators, respectively, then there is a function mapping $\theta : \mathcal{L} \rightarrow \mathcal{P} \cup \{\bot\}$, which associates a location $l \in \mathcal{L}$ with a location operator $\rho \in \mathcal{P}$, i.e. $\theta(l) = \rho$, or leave locations undefined as default, i.e. $\theta(l) = \bot$. Informally, it means that each location associates at most one location operator. For the sake of convenience, the function θ is called *location function*.

With location operators, each update multiset can be reduced to an update set by means of applying location operators over multisets of update values with respect to the corresponding locations. To be precise, we formalize the reduction as follows.

Definition 11. *Let $m = \{\!\!\{(l_1, t_1), ..., (l_n, t_n)\}\!\!\}$ be an update multiset, then there exists a reduced update set $\Box$ such that*

$$\Box = \{(l_i, b_i) | \theta(l_i) = \rho_i,\ m_i = \biguplus_{l_k = l_i} \{\!\!\{t_k\}\!\!\},\ \rho_i(m_i) = b_i,\ \rho_i \in \mathcal{P} \text{ and } i, k \in [1, n]\} \cup \{(l_i, t_i) | \theta(l_i) = \bot, \text{ and } i \in [1, n]\}$$

The question left is when an update multiset is reduced to an update set during computations. To answer this question, we first need to clarify the purpose of bringing in location operators. As illustrated in several scenarios of [6], various kinds of information can be generated during parallel computations, and furthermore, they may interact with each other in different manners. In practice, especially in the context of complex value databases, it is common that a collection of pieces of structures are retrieved from the database, then reconstructed as a whole to be an update value of a location. Let us have a look at the following example.

Example 1. Consider the tree structures shown in Figure 1. Suppose that we need to find out all names of people involved in the project titled "ABC" of $tree_1$, and reconstruct them into $tree_2$. The desired procedure of the computation is as follows. The names of persons are first retrieved out as update values corresponding to the same location. Then a location operator calculates over the set of names of persons, and returns a single update value, in which all names are grouped under a *Person* vertex. Finally, $tree_2$ is obtained by applying an update set consisting of that update and other updates deleting the original *Person* vertices on $tree_1$.

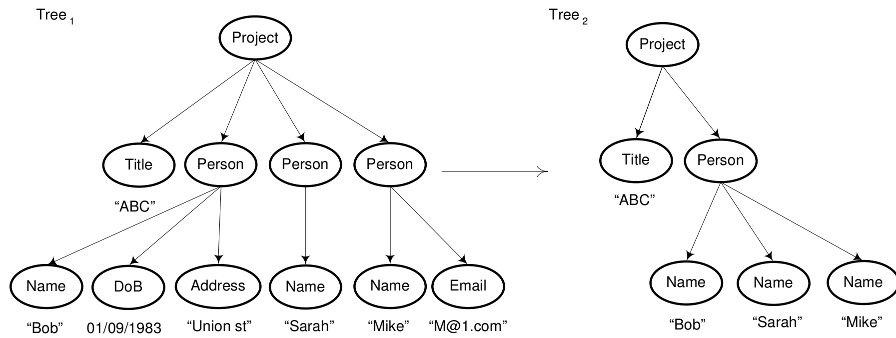

Fig. 1.

The above example shows that with a set of updates, a further computation over such updates might be in demand to construct the structures as required. To handle this kind of computations arising in the context of complex value databases in a general but simple manner, we propose the notion of location operator based on a key observation that parallel computations in our consideration are only caused by the evaluation of non-ground terms over a state, and thus a uniform manipulation during parallel computations associated with a non-ground term can always be formalized by a location operator over a set of locations associated with such a non-ground term. Furthermore, the possible

finite nestings of parallel computations yield the principle that location operators should have their scopes during nestings, and a reduction from an update multiset to an update set by applying location operators over update values is executed at the time when a nesting of parallel computations is finished. In order to obtain a better understanding, let us consider an example from [6].

Example 2. Consider the evaluation of a Boolean formula $\forall x \in D_1 \exists y \in D_2 \varphi(x,y)$.

Assume that the evaluation result will be stored at term t, and the cardinalities of elements in D_1 and D_2 are n_1 and n_2, respectively, then there are two nested parallel computations involved during the evaluation. At the inner parallel computations, with respect to each specific value $u_i \in D_1$ ($i \in [1, n_1]$) of x, there are n_2 parallel processes, each of which corresponds to term $\varphi(u_i, v_j)$, where $v_j \in D_2$ ($j \in [1, n_2]$), and produces an update $(t, true)$ or $(t, false)$, and $\theta(t) = \bigvee$. Thus the update multiset corresponding to the inner parallel computation contains the number n_2 of such updates, and then they are reduced to one update after applying $\bigvee$ over all update values occurring in the multiset since all of them have the same location. Afterwards, that update, which may be $(t, true)$ or $(t, false)$ with location operator $\theta(t) = \bigwedge$ specified by outer parallel computations, enter into the update multiset corresponding to the outer parallel computation which should have the number n_1 of such updates. Finally, that update multiset is reduced to be the update $(t, true)$ or $(t, false)$, which eventually make effects on the state by assigning term t with a truth value.

Now we need to define the consistency of an update set. One thing needs to be emphasized is about locations, in a general sense, which are defined over complex value structures. To illustrate this, we provide the following example.

Example 3. Consider structured updates on trees in Figure 2 and Figure 3. Let $\square_1$ and $\square_2$ be two update sets corresponding to Figure 2 and Figure 3 such that $\square_1 = \{(a\langle b\rangle, c)\}$ and $\square_2 = \{(a\langle e\rangle, d)\}$, where $a\langle b\rangle$ and $a\langle e\rangle$ indicate that vertex a is the parent of vertices b and e, respectively (For simplicity, we skip the related definition for these terms in this paper). By applying $\square_1$ on tree t, we can obtain tree t_1, and similarly, tree t_2 is obtained by applying $\square_2$ on tree t. How about the case that we apply an update set $\square = \square_1 \cup \square_2 = \{(a\langle b\rangle, c), (a\langle e\rangle, d)\}$ on tree t ? Although two locations in $\square$ are different, there still exists a clash in the update set $\square$.

The underlying reason for such a clash arising from the above example is that atomicity of locations in the context of complex value structures cannot be guaranteed in general. Indeed, with the popularity of the eXtensible Markup Language (XML), structured tree updates have been considered as an interesting research issue [8, 9]. To avoid possible clashes between locations, we introduce the notion of *separate update*. Here the notation $s \oplus \square$ denotes a new state generated by applying an update set $\square$ on a state s.

Definition 12. *Let* (l_1, t_1) *and* (l_2, t_2) *be two updates, and* s *be a state, then* (l_1, t_1) *and* (l_2, t_2) *are separate w.r.t. state* s *iff* $s_1 = s_2$ *holds, where*

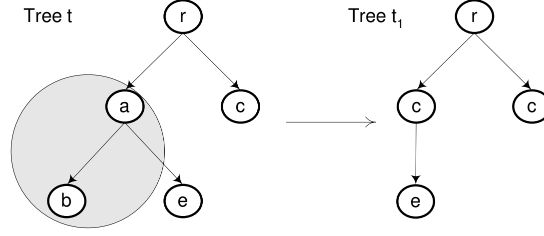

Fig. 2.

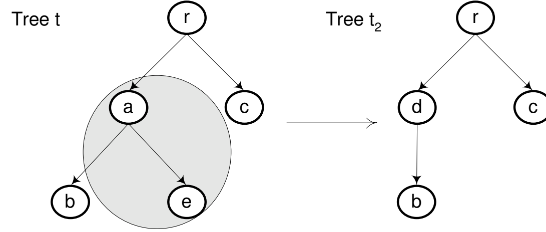

Fig. 3.

- $s_1 = (s \oplus \{(l_1, t_1)\}) \oplus \{(l_2, t_2)\}$, *and*
- $s_2 = (s \oplus \{(l_2, t_2)\}) \oplus \{(l_1, t_1)\}$.

It is easy to see that any two updates that have the identical location but different update values are not separate. The following separate update postulate describes the consistency of an update set in database transformations.

Definition 13. ***(separate update postulate).*** *Let* $\Pi = (S_\Pi, \tau_\Pi)$ *be a database transformation, then for any update set* $\Box$ *produced by* τ_Π *over a state* s,

- $\Box$ *is consistent iff all updates in* $\Box$ *are pairwise separate w.r.t. state* s.
- *Otherwise,* $\Box$ *is inconsistent.*

A state $s^{'} = s \oplus \Box$ *satisfying* $(s, s^{'}) \in \tau_\Pi$ *is obtained if* $\Box$ *is consistent, or a state* $s^{'} = s$ *satisfying* $(s, s^{'}) \in \tau_\Pi$ *is obtained if* $\Box$ *is inconsistent.*

An update set is *trivial* iff for each update in the update set, its location content is the same with its update value.

Lemma 1. *Let* $\Pi = (S_\Pi, \tau_\Pi)$ *be a database transformation, then*

- *for two states* $s_1, s_2 \in S_\Pi$ *satisfying* $(s_1, s_2) \in \tau_\Pi^n$, *there exist up to two non-trivial consistent update sets* $\Box_1$ *and* $\Box_2$ *such that* $s_2 = (s_1 \oplus \Box_1) \oplus \Box_2$, *and*
- *for two states* $s_1, s_2 \in S_\Pi$ *satisfying* $(s_1, s_2) \in \tau_\Pi$, *there exists up to one non-trivial consistent update set* $\Box_1$ *such that* $s_2 = s_1 \oplus \Box_1$.

Proof. (Sketch) For any two states $s_1, s_2 \in S_\Pi$ satisfying $(s_1, s_2) \in \tau_\Pi^n$, they have the same base set, but probably different dispose, active and reserve domains. Recall that the alteration of elements amongst domains must follow the unidirectional flow as shown in Subsection 2.2. If $Ddom(s_2) \cap Rdom(s_1) \neq \emptyset$, then two non-trivial consistent sets of updates are needed such that $s_2 = (s_1 \oplus \Box_1) \oplus \Box_2$. If $Ddom(s_1) \cap Rdom(s_2) = \emptyset$ and $s_1 \neq s_2$, then there exists exactly one non-trivial consistent update set such that $s_2 = s_1 \oplus \Box_1$. If $s_1 = s_2$, no non-trivial consistent update set exists.

If two states s_1 and s_2 satisfy $(s_1, s_2) \in \tau_\Pi$, then $Ddom(s_2) \cap Rdom(s_1) = \emptyset$ must hold due to the separate update postulate. Hence, up to one non-trivial consistent update set is needed to get $s_2 = s_1 \oplus \Box_1$.

2.6 Equivalent Structure Postulate

As specified in the abstract state postulate, databases transformations in our discussion are generic in the sense that the focus of database transformations is to capture structural properties of databases, instead of explaining atomic value. For this reason, all terms mentioned here are not ground terms. In terms of a database, some structures are indistinguishable with non-ground terms in general. For clarity, we can have a look at the following example, in which tuples (a, b, c) and (d, e, f) of the relation are indistinguishable.

A_1	A_2	A_3
a	b	c
d	e	f
g	k	k

In fact, classes of indistinguishable structures can exist among structures under any kind of data models. The common feature is that they can be arbitrarily exchanged within a database without any side effects on structures since there is no way to distinguish them. To facilitate the formalization, we use the notations $el(Y)$ to denote all elements occurring in structure Y and f^Y to denote the interpretation of function name f in structure Y. Let Y_1 and Y_2 be two structures, then Y_1 is said to be a *substructure* of Y_2, denoted by $Y_1 \preceq Y_2$, iff the following conditions are satisfied.

- $el(Y_1) \subseteq el(Y_2)$, and
- for every n-arity function name f, $f^{Y_1} = f^{Y_2}|_{el(Y_1)^n}$, where $f^{Y_2}|_{el(Y_1)^n}$ means the restriction of f^{Y_2} on domain $el(Y_1)^n$.

Suppose that Y is the structure of a state s, then any two substructures $Y_1, Y_2 \preceq Y$ are *equivalent* (denoted by $Y_1 \equiv Y_2$) iff exchanging them gives rise to an automorphism of Y.

Definition 14. ***(equivalent structure postulate).*** *Let Π= (S_Π, τ_Π) be a database transformation, then for any state $s_i \in S_\Pi$ whose structure contains*

a set $\{E_1, ..., E_n\}$ *of equivalence classes of substructures, the structure* Y_{i+1} *of state* s_{i+1} *such that* $(s_i, s_{i+1}) \in \tau_\Pi$ *must satisfy the following condition for each equivalence class* E_j $(j \in [1, n])$*.*

- *For any* $Y \in E_j$ *satisfying* $Y \preceq Y_{i+1}$*, it implies that for every permutation* σ *of* E_j*,* $\sigma(Y) \preceq \sigma^{'}(Y_{i+1})$*, where* $\sigma^{'}(Y_{i+1})$ *is the structure of state* $s^{'}_{i+1}$ *satisfying* $(s_i, s^{'}_{i+1}) \in \tau_\Pi$*, and* $\sigma^{'}$ *is a unique isomorphism of* Y_{i+1} *into* $\sigma^{'}(Y_{i+1})$ *that extends* σ*.*

Note that, in contrast to classes of indistinguishable object discussed in [2], which is defined over two objects that can not be distinguished by any relational machine with k variables, our definition of equivalence relation is more strict, as it requires that two structures can not be distinguished no matter how many variables are involved because they have identical structural properties. The key idea of the equivalent structure postulate is to guarantee a fair treatment for every member of an equivalence class during one-step transformations.

The following two lemmata are obtained on the basis of the equivalent structure postulate, and they will be helpful for the proof of our main result.

Lemma 2. *For a database transformation* $\Pi = (S_\Pi, \tau_\Pi)$*, if a structure* Y *in an equivalence class* E *of a state* $s_i \in S_\Pi$ *is a substructure of the structure of every state* s_{i+1} *satisfying* $(s_i, s_{i+1}) \in \tau_\Pi$*, then the structure of every state* s_{i+1} *must have all structures in* E *as substructures.*

Proof. We assume that there are k_1 states in $\{s_{i+1} | (s_i, s_{i+1}) \in \tau_\Pi\}$, and for clarity, they are denoted as $s_{i+1,1}, ..., s_{i+1,k_1}$, respectively. Besides, we also assume that there are k_2 elements in E, and they can be expressed by different permutations $\sigma_1, ..., \sigma_{k_2}$ of E on Y such that $\sigma_1(Y) = Y$ and $\sigma_n(Y) \neq Y$ $(n \in [2, k_2])$. Since Y is a substructure of structures of all states $s_{i+1,j}$ $(j \in [1, k_1])$, by using the equivalent structure postulate, any other structure $\sigma_n(Y)$ must be a substructure of structures of some states $s_{i+1,j}$. To get the contradiction, we assume that there exists a state $s_{i+1,a}$, which has Y but no $\sigma_n(Y)$. According to the equivalent structure postulate again, there must exist another state $s_{i+1,b}$ being isomorphic to state $s_{i+1,a}$, and $s_{i+1,b}$ has $\sigma_n(Y)$ but no Y. This contradicts the fact that Y is a substructure of structures of all states $s_{i+1,j}$.

Lemma 3. *For a database transformation* $\Pi = (S_\Pi, \tau_\Pi)$*, if a structure* Y *in an equivalence class* E *of a state* $s_i \in S_\Pi$ *is a substructure of the structure of some state* s_{i+1} *satisfying* $(s_i, s_{i+1}) \in \tau_\Pi$*, then the structure of that state* s_{i+1} *either has all structures in* E *as substructures, or associates a set of structures of states* $s^{'}_{i+1}$ *satisfying* $(s_i, s^{'}_{i+1}) \in \tau_\Pi$ *, which is closed under the permutation of* E*.*

Proof. It is easy to prove that, the cases that the structure of state s_{i+1} have all structures in E, and that the structure of state s_{i+1} associates a set of structures of states $s^{'}_{i+1}$ satisfying $(s_i, s^{'}_{i+1}) \in \tau_\Pi$, which is closed under the permutation of E, satisfy the equivalent structure postulate. The rest of proof is to show that only these two cases can exist in state s_{i+1}. We assume that there are k_2 elements in E, and k_1 states in $\{s_{i+1} | (s_i, s_{i+1}) \in \tau_\Pi\}$ denoted as $s_{i+1,1}, ..., s_{i+1,k_1}$,

respectively. Let $a, b_1, ..., b_n \in [1, k_1]$. To derive the contradiction, we further assume that state s_i has $Y_1, ..., Y_n$ $(n < k_2)$ as its substructures and there does not exist a state $s_{i+1,a}$, which has $\sigma(Y_1), ..., \sigma(Y_n)$ as its substructures, where σ denotes a permutation of E. In accordance with the equivalent structure postulate, since s_i has Y_1, there must exist a state s_{i+1,b_1} which has $\sigma(Y_1), Y_2..., Y_n$ as its substructures. By inductively following the same argument, we know that there must exist a state s_{i+1,b_2} having $\sigma(Y_1), \sigma(Y_2), Y_3, ..., Y_n$ as its substructures,..., and a state s_{i+1,b_n} having $\sigma(Y_1), \sigma(Y_2), ..., \sigma(Y_n)$ as its substructures. Apparently, s_{i+1,b_n} contradicts $s_{i+1,a}$ in our assumption.

3 Abstract Database Transformation Machines

We define a notion of Abstract Database Transformation Machine (ADTM) in this section, by presenting the terms and rules of ADTMs.

Definition 15. *An Abstract Database Transformation Machine (ADTM) Λ is a pair $(S_\Lambda, \tau_\Lambda)$, where*

- *S_Λ is a set of states with finite structures of a fixed signature H, and*
- *τ_Λ is an abstract database transformation program.*

Each state $s \in S_\Lambda$ has two spaces: *computation space* and *database space.* In addition, signature H has two disjoint sets of function names in computation and database spaces, denoted by $\mathfrak{F}_C$ and $\mathfrak{F}_D$, correspondingly. For an ADTM Λ, its terms consist of *computation terms* and *database terms*, which are separately defined over signature H.

Let $\mathbf{D}$ be the domain of computation space, $\mathcal{X}_C \subseteq \mathfrak{F}_C$ be a set of nullary function names (i.e., variables) and $\mathcal{F}_C \subseteq \mathfrak{F}_C$ be a set of non-nullary function names, then a set $\mathcal{T}_C$ of computation terms is defined by

- $\mathbf{D} \subseteq \mathcal{T}_C$,
- $\mathcal{X}_C \subseteq \mathcal{T}_C$, and
- $f(t_1, ...t_n) \in \mathcal{T}_C$, where $f \in \mathcal{F}_C$ and $t_i \in \mathcal{T}_C$ $(i \in [1, n])$.

Let $\mathcal{X}_D \subseteq \mathfrak{F}_D$ be a set of nullary function names and $\mathcal{F}_D \subseteq \mathfrak{F}_D$ be a set of non-nullary function names, then a set $\mathcal{T}_D$ of database terms is defined by

- $\mathcal{X}_D \subseteq \mathcal{T}_D$, and
- $f(t_1, ...t_n) \in \mathcal{T}_D$, where $f \in \mathcal{F}_D$ and $t_i \in \mathcal{T}_D$ $(i \in [1, n])$.

For convenience, we use the notations $\Delta^Y(r)$ and $\ddot{\Delta}^Y(r)$ refer to an update set and an update multiset produced by executing the rule $r \in \mathcal{R}$ on the structure Y, respectively, and the notation $val^Y(t)$ to denote the evaluation of a term t over structure Y.

Given a state, the computation and database terms are evaluated over its computation and database space, respectively. More precisely, the evaluation $val^Y(t)$ of a term $t \in \mathcal{T}_C$ produces a substructure of Y in computation space, whereas the evaluation $val^Y(t)$ of a term $t \in \mathcal{T}_D$ results in a set of substructures of Y in database space. A term t is a Boolean term iff $val^Y(t) \subseteq \{true, false\}$.

Definition 16. *Let $\mathcal{T} = \mathcal{T}_C \cup \mathcal{T}_D$ and $\mathcal{X} = \mathcal{X}_C \cup \mathcal{X}_D$, then for an Abstract Database Transformation Machine $\Lambda = (S_\Lambda, \tau_\Lambda)$ with a signature H, τ_Λ is inductively defined by a set $\mathcal{R}$ of rules.*

- *update rule:*

$$f(t_1, ...t_n) \ := t_0,$$

  where $f(t_1, ...t_n)$ and t_0 are terms in $\mathcal{T}$. An update rule executed at a structure Y with signature H produces an update $(f(a_1, ...a_n), b)$, where $a_i \in val^Y(t_i)$ $(i \in [1, n])$ and $b \in val^Y(t_0)$. Therefore, $\Delta^Y(f(t_1, ...t_n) \ := t_0) = \{(f(a_1, ...a_n), b)\}$.
- *conditional rule:*

$$\text{if } t \text{ then } r,$$

  where $t \in \mathcal{T}$ is a Boolean term, and $r \in \mathcal{R}$ is a rule. A conditional rule executed at a structure Y with signature H produces an update set $\Delta^Y(\text{if } t \text{ then } r) = \Delta^Y(r)$ iff $val^Y(t) = \{true\}$. Otherwise, $\Delta^Y(\text{if } t \text{ then } r) = \emptyset$.
- *forall rule:*

$$\text{forall } x \text{ with } \varphi(x) \text{ do } r,$$

  where $x \in \mathcal{X}$ is a variable, $\varphi(x) \in \mathcal{T}$ is a Boolean term containing x, and $r \in \mathcal{R}$ is a rule. A forall rule executed at a structure Y with signature H first produces an update multiset $\ddot{\Delta}^Y(\text{forall } x \text{ with } \varphi(x) \text{ do } r) = \biguplus_{i \in [1,n]} \ddot{\Delta}^Y([a_i/x]r)$, where $\{x | val^Y(\varphi(x)) = \{true\}\} = \{a_1, ..., a_n\}$, and $[a_i/x]r$ means that variable x is bounded to value a_i within the rule r. Then an update set $\Delta^Y(\text{forall } x$ with $\varphi(x) \text{ do } r)$ is obtained by applying location operators on update values that have the same locations in the multiset $\ddot{\Delta}^Y(\text{forall } x \text{ with } \varphi(x) \text{ do } r)$ as defined in Definition 11.
- *choose rule:*

$$\text{choose } x \text{ with } \varphi(x) \text{ do } r,$$

  where $x \in \mathcal{X}$ is a variable, $\varphi(x) \in \mathcal{T}$ is a Boolean term containing x, and $r \in \mathcal{R}$ is a rule. A choose rule executed at a structure Y with signature H produces an update set $\Delta^Y(\text{choose } x \text{ with } \varphi(x) \text{ do } r) = \Delta^Y([a/x]r)$, where $a \in \{x | val^Y(\varphi(x)) = \{true\}\}$, and $[a/x]r$ means that variable x is bounded to value a within the rule r.
- *parallel rule:*

$$\text{par } r_1 \ r_2 \text{ par},$$

  where $r_1, r_2 \in \mathcal{R}$ are rules. A parallel rule executed at a structure Y with signature H produces an update set $\Delta^Y(\text{par } r_1 \ r_2 \text{ par}) = \Delta^Y(r_1) \cup \Delta^Y(r_2)$.
- *sequence rule:*

$$\text{seq } r_1 \ r_2 \text{ seq },$$

where $r_1, r_2 \in \mathcal{R}$ *are rules. A sequence rule executed at a structure* Y *with signature* H *produces an update set* $\Delta^Y(\text{seq } r_1 \ r_2 \text{ seq}) = \Delta^Y(r_1) \oslash \Delta^Y(r_2)$*, where* $\Delta^Y(r_1) \oslash \Delta^Y(r_2)$ *means an update set equivalent to the result by first applying* $\Delta^Y(r_1)$ *on the structure* Y*, and then applying* $\Delta^Y(r_2)$ *on the resulting structure.*

– *let rule:*

$$\text{let } \theta(t) = \rho \text{ in } r,$$

where θ *is a location function,* $t \in \mathcal{T}$ *is a term.* $\rho \in \mathcal{P}$ *is a location operator and* $r \in \mathcal{R}$ *is a rule. A let rule executed at a structure* Y *with signature* H *produces an update set* $\Delta^Y(\text{let } \theta(t) = \rho \text{ in } r) = \Delta^Y([\rho/\theta(l)]r)$*, where* $l \in \{x | x \in val^Y(t)\}$*, and* $[\rho/\theta(l)]r$ *means that the location operator associated with* l *is bounded to* ρ *within the rule* r*.*

Compared with the rules of ASMs, the rules of ADTMs are defined as usual, except for the let rule. In fact, the let rule is modified to provide an enhanced functionality assigning location operators to locations. Let us have a look at the following example, which is similar to the one provided in [6].

Example 4. Consider the task to count the number of tuples within a unary relation *title*. The following rule r can be written down to get the desired result.

forall x *with* $title(x)$ *do*
  let $\theta(t) = \Sigma$ *in*
    $t := 1$

Σ represents the aggregation function *sum*, and is assigned to be the location operator of the location t. The rule r is executed by two steps. First of all, an update multiset $\ddot{\triangle}(r) = \{\!\{(t,1), \dots, (t,1)\}\!\}$ is generated. Secondly, all update values of t (i.e., that is always 1 in this case.) are summed up by applying the location operator Σ of t. Hence, the number of tuples in relation *title* is obtained, which is the same with the cardinality of $(t,1)$ occurring in $\ddot{\triangle}(r)$.

In addition, we do not introduce the import rule as the import rule can be easily simulated by using the choose rule and a specific unary function *new*.

Theorem 2. *Every Abstract Database Transformation Machine represents a database transformation.*

Proof. (Sketch) To prove the above Theorem, we need to check whether an Abstract Database Transformation Machine satisfies all the postulates of a database transformation defined in Section 2. Since there is no any iteration involved in the set $\mathcal{R}$ of rules, and furthermore the set $\mathcal{T}$ of terms reflects higher-order structures associated with a fixed and finite signature, it is easy to see that an ADTM satisfies the finite sequence and abstract state postulates. The data abstract postulate can also be satisfied as its purpose is to provide necessary

information related to the data abstraction of states. Besides, the finite length of an abstract database transformation program and the evaluation of terms on finite structures determine that an ADTM satisfies the exploration boundary postulate. The separate update postulate is satisfied by specifying location operators and reductions from update multisets to update sets, and then checking locations in update sets of ADTMs. The equivalent structure postulate is guaranteed by the separation of computation terms and database terms, where database terms are evaluated over abstract database spaces in ADTMs.

4 Characterization Theorem

In this section, we prove the main result of this paper that Abstract Database Transformation Machines can behaviourally simulate any database transformation over complex value structures, which is the converse of the previous theorem. This result will follow from a series of lemmata.

Lemma 4. *Given a database transformation $\Pi = (S_\Pi, \tau_\Pi)$, for each state s with the structure Y satisfying $s \in S_\Pi$ and $s \notin F_\Pi$, there exists a rule $r \in \mathcal{R}$ of ADTMs such that*

$$\{s^{'}|(s,s^{'}) \in \tau_\Pi\} = \{s^{''}|s^{''} = s \oplus \Delta^Y(r)\}.$$

Proof. (Sketch)

By following the exploration boundary postulate, we assume that the cardinality of set $\{s^{'}|(s,s^{'}) \in \tau_\Pi\}$ is k. In accordance with Lemma 1, for all states in $\{s^{'}|(s,s^{'}) \in \tau_\Pi\}$ we can use $\Box_1, ..., \Box_k$ to denote their update sets with respect to state s, respectively. The construction of r consists of two parts, which capture the common updates among $\Box_1, ..., \Box_k$, and the rest of them, correspondingly.

First of all, according to the equivalent structure postulate and Lemma 2, for all locations L_1 occurring in updates of $\bigcap\limits_{u\in[1,k]} \Box_u$, we can always find a set $\{E_1, ..., E_a\}$ of equivalence classes satisfying $L_1 = E_1 \cup ... \cup E_a$. Then, based on these equivalence classes and locations, updates in $\bigcap\limits_{u\in[1,k]} \Box_u$ can be partitioned into groups. Each group corresponds to an equivalent class and can be captured by an update rule. Moreover, all update rules for the groups can be executed in parallel following from the separate update postulate. That is,

par
  $forall\ x_{10}, ..., x_{1i}\ with\ \varphi(x_{10}, ...x_{1i})\ do$
    $let\ \theta(f_1(x_{11}, ...x_{1i})) = \rho_1\ in$
      $f_1(x_{11}, ...x_{1i}) := x_{10}$

  $forall\ x_{20}, ..., x_{2j}\ with\ \varphi(x_{20}, ...x_{2j})\ do$
    $let\ \theta(f_2(x_{21}, ...x_{2j})) = \rho_2\ in$
      $f_2(x_{21}, ...x_{2j}) := x_{20}$
  ...

par

To be precise, let us use r_1 to refer to the above rule, then $\triangle^Y(r_1) = \bigcap_{u\in[1,k]} \Box_u$.

Secondly, we consider the rest of updates in $\Box_1, ..., \Box_k$, which can be expressed as $\Box_1 - \bigcap_{u\in[1,k]} \Box_u, ..., \Box_k - \bigcap_{u\in[1,k]} \Box_u$, respectively. Similarly, according to the equivalent structure postulate and Lemma 3, for all locations L_2 occurring in updates of $\bigcup_{u\in[1,k]} \Box_u - \bigcap_{u\in[1,k]} \Box_u$, we can always find a set $\{E_1^{'}, ..., E_b^{'}\}$ of equivalence classes satisfying $L_2 = E_1^{'} \cup ... \cup E_b^{'}$. Furthermore, following Lemma 3, it is straightforward to conclude that with respect to an equivalent class $E_i^{'}$ ($i \in [1, b]$) there are three possible situations occurred in an update set $\Box_j - \bigcap_{u\in[1,k]} \Box_u$ ($j \in [1, k]$). I.e.,

- all of structures of $E_i^{'}$ are its locations, or
- none of structures of $E_i^{'}$ is its location, or
- some of structures of $E_i^{'}$ are its locations, and it associates a set of update sets $\Box_j - \bigcap_{u\in[1,k]} \Box_u$, which is closed under a permutation of $E_i^{'}$.

Therefore, the rest of updates in $\Box_1, ..., \Box_k$ corresponds to the following choose rules executed in parallel, in which each outermost choose rule deals with an equivalent class in a general manner.

```
par
 choose x1 with ψ11(x1) ∨ ψ12(x1) ∨ ψ13(x1) do
   if ψ11(x1) then
     choose y1', y11, ..., y1p with φ(y1', y11, ..., y1p) do
       f1'(y11, ..., y1p) := y1'
     choose y1'', y11, ..., y1p with φ(y1'', y11, ..., y1p) do
       f1'(y11, ..., y1p) := y1''
     ......
   if ψ12(x1) then
     forall y10, ..., y1p with φ(y10, ..., y1p) do
       let θ(f1'(y11, ...y1p)) = ρ1' in
         f1'(y11, ..., y1p) := y10

 choose x2 with ψ21(x2) ∨ ψ22(x2) ∨ ψ23(x2) do
   if ψ21(x2) then
     choose y2', y21, ..., y2q with φ(y2', y21, ..., y2q) do
       f2'(y21, ..., y2q) := y2'
     choose y2'', y21, ..., y2q with φ(y2'', y21, ..., y2q) do
       f2'(y21, ..., y2q) := y2''
     ......
   if ψ22(x2) then
```

```
    forall y20,...,y2q with φ(y20,...,y2q) do
      let θ(f'2(y21,...y2q)) = ρ'2 in
        f'2(y21,...,y2q) := y20
...
par
```

Again, let us use r_2 to refer to the above rule, then $\{\triangle^Y(r_2)\} = \{\Box_i - \bigcap_{u\in[1,k]} \Box_u | i \in [1,k]\}$. Note that, indeed, only consistent update sets generated by r_2 are of our interest since all inconsistent update sets have the same effects on state s as empty sets. Therefore, a general form of the desirable rule r is to combine the above two parts into the following.

```
par
  forall x10,...,x1i with φ(x10,...x1i) do
    let θ(f1(x11,...x1i)) = ρ1 in
      f1(x11,...x1i) := x10
  forall x20,...,x2j with φ(x20,...x2j) do
    let θ(f2(x21,...x2j)) = ρ2 in
      f2(x21,...x2j) := x20
  ...
  choose x1 with ψ11(x1) ∨ ψ12(x1) ∨ ψ13(x1) do
    if ψ11(x1) then
      choose y'1,y11,...,y1p with φ(y'1,y11,...,y1p) do
        f'1(y11,...,y1p) := y'1
      choose y''1,y11,...,y1p with φ(y''1,y11,...,y1p) do
        f'1(y11,...,y1p) := y''1
      ......
    if ψ12(x1) then
      forall y10,...,y1p with φ(y10,...,y1p) do
        let θ(f'1(y11,...y1p)) = ρ'1 in
          f'1(y11,...,y1p) := y10

  choose x2 with ψ21(x2) ∨ ψ22(x2) ∨ ψ23(x2) do
    if ψ21(x2) then
      choose y'2,y21,...,y2q with φ(y'2,y21,...,y2q) do
        f'2(y21,...,y2q) := y'2
      choose y''2,y21,...,y2q with φ(y''2,y21,...,y2q) do
        f'2(y21,...,y2q) := y''2
      ......
    if ψ22(x2) then
      forall y20,...,y2q with φ(y20,...,y2q) do
        let θ(f'2(y21,...y2q)) = ρ'2 in
          f'2(y21,...,y2q) := y20
  ...
par
```

Lemma 5. *Let* $\Pi = (S_\Pi, \tau_\Pi)$ *be a database transformation, for all states* s_n *with structures* Y *satisfying* $(s_0, s_n) \in \tau_\Pi^n$, $s_n \in S_\Pi$ *and* $s_n \notin F_\Pi$, *there exists the same rule* $r \in \mathcal{R}$ *of ADTMs such that*

$$\{s_{n+1} | (s_n, s_{n+1}) \in \tau_\Pi\} = \{s'_{n+1} | s'_{n+1} = s_n \oplus \Delta^Y(r)\}.$$

Proof. By applying the equivalent structure postulate, we can assume that the cardinality of set $\{s_n | (s_0, s_n) \in \tau_\Pi^n\}$ is d, and all states in $\{s_n | (s_0, s_n) \in \tau_\Pi^n\}$ are thus expressed as $s_{n,1}, ..., s_{n,d}$, respectively. Since all states in $\{s_n | (s_0, s_n) \in \tau_\Pi^n\}$ must be distinct from each other, and also the signature is fixed according to the abstract state postulate, there exists a set of Boolean terms $\{t_1, ..., t_d\}$, in which a term t_i ($i \in [1, d]$) can only be evaluated to be true at the structure of state $s_{n,i}$. Note that the terms can be constructed over complex value structures satisfying the data abstraction postulate. Therefore, based on a set of rules $\{r_1, \ldots, r_d\}$ obtained from Lemma 4, in which a rule r_i ($i \in [1, d]$) satisfies $\{s_{n+1} | (s_{n,i}, s_{n+1}) \in \tau_\Pi\} = \{s'_{n+1} | s'_{n+1} = s_{n,i} \oplus \Delta^Y(r_i)\}$, we can construct the desirable rule r as follows:

par
  if t_1 *then* r_1
  ...
  if t_d *then* r_d
par

Lemma 6. *Let* $\Pi = (S_\Pi, \tau_\Pi)$ *be a database transformation,* n *be the arity of the run relation of* Π, *and* Y *be the structure of the initial state* s_0, *then there exists a rule* $r \in \mathcal{R}$ *of ADTMs satisfying* $\{s_k | (s_0, s_k) \in \tau_\Pi^k\} = \{s'_k | s'_k = s_0 \oplus \Delta^Y(r)\}$, *for any* $k \in [1, n]$.

Proof. The proof is quite straightforward. There are two steps. Firstly, by applying Lemma 5, a set $\{r_1, .., r_n\}$ of rules can be constructed, in which each r_i ($i \in [1, n]$) satisfies $\{s_i | (s_{i-1}, s_i) \in \tau_\Pi\} = \{s'_i | s'_i = s_{i-1} \oplus \Delta^{Y'}(r_i)\}$ for all states s_{i-1} of $(s_0, s_{i-1}) \in \tau_\Pi^{i-1}$, where Y' denotes the structures corresponding to states s_{i-1}. Secondly, the rule r can be simulated by using the sequence rule. That is

$$seq \;\; r_1 \;\; \ldots \;\; r_n \;\; seq.$$

Theorem 3. *For every database transformation* $\Pi = (S_\Pi, \tau_\Pi)$, *there exists a behaviourally equivalent Abstract Database Transformation Machine* $\Lambda = (S_\Lambda, \tau_\Lambda)$.

Proof. On the basis of Lemmata 4, 5 and 6, it can be conclude that

- $s_0^\Lambda = s_0^\Pi$,
- $F_\Lambda = F_\Pi$,
- $S_\Lambda = S_\Pi$, and
- $\tau_\Lambda = \tau_\Pi$,

where $s_0^{\Pi} \in S_{\Pi}, s_0^{\Lambda} \in S_{\Lambda}$ are initial states of Π and Λ, respectively, and $F_{\Pi} \subseteq S_{\Pi}, F_{\Lambda} \subseteq S_{\Lambda}$ are sets of final states of Π and Λ, respectively.

Finally, according to Definition 5, it can be concluded that every database transformation $\Pi = (S_{\Pi}, \tau_{\Pi})$ can be behaviourally simulated by an Abstract Database Transformation Machine $\Lambda = (S_{\Lambda}, \tau_{\Lambda})$ because of $s_0^{\Lambda} = s_0^{\Pi}$, $F_{\Lambda} = F_{\Pi}$, $S_{\Lambda} = S_{\Pi}$, and $\tau_{\Lambda} = \tau_{\Pi}$.

Remark 1. In the preceding proof of Theorem 3, the sequence rule is used in Lemma 6 to simulate the sequences of states associated with runs. However, it is also possible to prove Theorem 3 without using the sequence rule. Limited to the space, we skip an alternative approach in this paper.

5 Conclusions

In this paper, we first have examined the postulates stipulated on database transformations over complex value structures. Following this, the notion of Abstract Database Transformation Machine has been proposed, and furthermore we showed that ADTMs can simulate any database transformation over complex value structures satisfying the postulates. The main contributions of this paper are the following.

- We proposed a computation model, which generalizes not only queries but also updates of database transformations into a unified framework.
- The computation model provides a strong capability to manipulate complex value structures by taking into consideration higher-order structures.
- The parallel computations involved in database transformations are handled by using a simple and elegant approach, in which location operators are associated with locations.

In addition, we notice that a class of database transformations that may lead to an infinite sequence of states within a run has failed to be captured by the proposed computation model. For instance, a run can be an infinite sequence ending with non-terminating regular alternation amongst several states. In another case, a run might have states of infinite structures, which can be formalized with finite descriptions. The extension in these directions will be addressed in our future work.

References

1. Abiteboul, S., and Kanellakis, P. C. Object identity as a query language primitive. In *SIGMOD '89: Proceedings of the 1989 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 1989), ACM Press, pp. 159–173.
2. Abiteboul, S., Papadimitriou, C. H., and Vianu, V. The power of reflective relational machines. In *Logic in Computer Science* (1994), pp. 230–240.

3. Abiteboul, S., and Vianu, V. Datalog extensions for database queries and updates. Tech. Rep. RR-0900, 09 1988.
4. Abiteboul, S., and Vianu, V. Generic computation and its complexity. In *STOC '91: Proceedings of the twenty-third annual ACM symposium on Theory of computing* (New York, NY, USA, 1991), ACM Press, pp. 209–219.
5. Beeri, C. A formal approach to object-oriented databases. *Data Knowl. Eng. 5*, 4 (1990), 353–382.
6. Blass, A., and Gurevich, Y. Abstract state machines capture parallel algorithms. *ACM Trans. Comput. Logic 4*, 4 (2003), 578–651.
7. Blass, A., Gurevich, Y., and Shelah, S. Choiceless polynomial time. Tech. Rep. CSE-TR-338-97, 16 1997.
8. Calcagno, C., Gardner, P., and Zarfaty, U. A context logic for tree update, 2004.
9. Calcagno, C., Gardner, P., and Zarfaty, U. Context logic and tree update. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2005), ACM Press, pp. 271–282.
10. Chen, P. P.-S. The entity-relationship model-toward a unified view of data. *ACM Trans. Database Syst. 1*, 1 (1976), 9–36.
11. Codd, E. F. A relational model of data for large shared data banks. *Commun. ACM 26*, 1 (1983), 64–69.
12. Cohen, S. User-defined aggregate functions: bridging theory and practice. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 2006), ACM Press, pp. 49–60.
13. Gurevich, Y. Sequential abstract-state machines capture sequential algorithms. *ACM Trans. Comput. Log. 1*, 1 (2000), 77–111.
14. Gurevich, Y., and Yavorskaya, T. On bounded exploration and bounded nondeterminism. Tech. Rep. MSR-TR-2006-07, Microsoft Research, January 2006.
15. Neven, F., and Schwentick, T. Automata- and logic-based pattern languages for tree-structured data, 2001.
16. Schewe, K.-D., and Thalheim, B. Fundamental concepts of object oriented databases. *Acta Cybern. 11*, 1-2 (1993), 49–84.
17. Van den Bussche, J. *Formal Aspects of Object Identity in Database Manipulation.* PhD thesis, University of Antwerp, 1993.
18. Van den Bussche, J., and Van Gucht, D. Semi-determinism (extended abstract). In *PODS '92: Proceedings of the eleventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems* (New York, NY, USA, 1992), ACM Press, pp. 191–201.
19. Van den Bussche, J., and Van Gucht, D. Non-deterministic aspects of object-creating database transformations. In *Selected Papers from the Fourth International Workshop on Foundations of Models and Languages for Data and Objects* (London, UK, 1993), Springer-Verlag, pp. 3–16.
20. Van Den Bussche, J., Van Gucht, D., Andries, M., and Gyssens, M. On the completeness of object-creating database transformation languages. *J. ACM 44*, 2 (1997), 272–319.
21. World Wide Web Consortium. Document object model (DOM) level 1 specification.